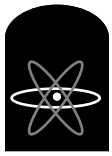
UCRL-ID-114566
NRC FIN L-1867

Real-Time Systems Complexity and Scalability

G.G. Preckshot

May 28, 1993



FESSP
Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

Disclaimer

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy.

Real-Time Systems
Complexity and Scalability

G.G. Preckshot

Manuscript Date: May 28, 1993

Contents

Abbreviations

COCOMO	COnstructive COst MOdel
CPU	Central Processing Unit
CSMA	Carrier Sensitive Multiple-Access
IEEE	Institute of Electrical and Electronics Engineers
MoD	Ministry of Defence (British)
NRC	Nuclear Regulatory Commission
PID	Proportional-Integral-Derivative
PLC	Programmable Logic Controllers
RADC	Rome Air Development Center
RISC	Reduced Instruction Set Computers
SCSI	Small Computer System Interface
SLOC	Source Lines of Code

Real-Time Systems
Complexity and Scalability

Executive Summary

This report is the culmination of a study to identify system and software complexity metrics that will assist the Nuclear Regulatory Commission (NRC) in evaluating a system design. Complexity is the degree to which a system or component has a design or implementation that is difficult to understand and verify. To be effective, complexity measures must evaluate the system and the software implemented within the system. Using current tools and methodologies, determining system and software complexity before the system is built is very difficult, if not impossible. There are many tools and techniques available to measure complexity *after* a system is built, but making changes to the hardware and software at this point is difficult and expensive. Measuring complexity *during* the building process is the best way to determine and reduce unnecessary complexity.

Most of the quantitative measures of complexity have been designed to measure software, and therefore constitute the majority of the complexity metrics identified in this report. Currently, research is still directed at software complexity and software quality measures. Evaluation of system complexity relies mainly on qualitative evaluations. Detection and correction of complexity problems can be achieved if the products of each phase of development are evaluated before going on to the next phase of development. This report presents an approach for characterizing system complexity based on the attributes of the architecture and software components.

This report focuses on complexity and scalability. However, there are additional factors that affect the ability of the system to meet safety and performance requirements. Even with a good set of system and software requirements, many errors can be injected into the software due to the lack of good software engineering practices, and may remain because of insufficient testing.

There is no simple solution to evaluating software and system complexity. A starting point is the use of good systems engineering and software engineering practices in the design and implementation of a system. For software, the use of complexity metrics to identify and eliminate high-complexity components is a first step to reduce risk. The most popular software metric is the McCabe complexity metric, which indicates the complexity of low-level software modules. Industry generally agrees on a value for this metric that indicates acceptable levels of complexity and development risk.

Continual evaluation of performance and complexity of systems and software during each phase of development will result in the early identification of problems and cost-effective solutions.

Based on available literature and experience, the report concludes that there are no widely used metrics for software or system complexity. Problems also exist with adapting many complexity metrics to practical systems. McCabe and Butler note that practical systems normally have 100 times the design complexity of those used in textbooks.

Achieving NRC's goal will require a research effort to define metrics and to determine values that will indicate an acceptable hazard level, although little evidence suggests a reliable link between hazard and complexity. A combination of metrics and development standards may give NRC a firmer foundation on which to evaluate and approve systems.

1. Introduction

Real-time computer systems are essential parts of many modern control or protection systems, yet there is no generally accepted way to predict system complexity and performance. Questions like, "Will it work if we make it bigger?" "Is it safe?" "Is it fast enough?" "Does it work correctly?" have no ready answer. It is generally conceded that as such systems get "bigger," they become less safe, go slower, and make more errors. Even the notion of "bigger" has been hard to define because real-time systems are composed both of hardware and software and, while hardware components are easy to count, software has proved much harder to measure. The interactions between hardware, software, and the real world are even more difficult to quantify.

At the root of our conceptual problems is system complexity, for which there is no facile description. In addition, software adds a level of complexity to the traditional hardware-only system. Complexity has been approached by attempting to measure its aspects and their effects upon system qualities of interest. Most research effort has been directed to measuring one component of real-time systems, the software, because this component has historically been the cause of most uncertainty in system cost, reliability, and performance. The current state of that research is covered in the body of the report.

While attempts to measure software have proceeded slowly, industry and the software engineering profession have had to design and build complex real-time systems without benefit of research results. Two decades of software practice and experience have produced a number of indicators that tell the practitioner that a design is more or less complex. From experience, most designers know system attributes that make development and verification more difficult and make systems less reliable. Some of these attributes are presented later with justification from the literature. If no precise complexity measures are available, complexity attributes may still give an idea, albeit less precise, of the level and types of complexity present in a real-time system. To this end, an unvalidated taxonomy of real-time systems is introduced, which permits a rough classification of such systems into seven complexity levels.

Rightly or wrongly, small systems are considered simple and big systems are considered complex. Closely related to the idea of complexity is the idea of scaling, changing the size of one or more aspects of a real-time system. Scaling a real-time system can have unexpected effects on system response, reliability, and correctness. Some of the areas which historically have caused problems are covered in the body of the report. Scalability issues are important when a particular real-time system is not precisely adapted to a proposed application, as would happen when applying a commercial generic real-time control system to an actual industrial plant.

1.1 Purpose

The purpose of this report is to support the NRC's primary goal to estimate the hazard level of a system design. If the hazard level is found to be unacceptably high, the NRC may proscribe certain types of complexity for safety systems. It is desirable to identify quantitative measures or define a qualitative marker list of computer hardware architecture complexity to support this goal.

1.2 Scope

The report provides a summary of approaches and measures used by industry to evaluate system and software complexity. It includes evaluation methodology available to measure complexity for both systems and software, and presents approaches to defining system complexity and classifying systems according to complexity attributes. A discussion of system scalability identifies risk factors in developing large systems.

2. Complexity

Complexity is an indicator of how difficult a system, program, or module is to understand and maintain (Arthur 1985). The more complex a system, program, or module, the harder it is to analyze, test, and maintain.

System complexity and software complexity must be considered when judging the validity of a system design. System and software complexity measures are most often used to determine how difficult a design or design change will be to make and how long it will take to complete. They can also be used as indicators of how well a system has been designed. Systems, programs, and modules with high complexity levels are more likely to be delivered with errors than less complex ones that are easier to understand, maintain, and test.

When developing any system, an early prediction of system complexity is important. Estimating system risk early in the life cycle gives adequate time to redesign overly complex aspects of the system. When reviewing a system concept to determine if building the system as defined is reasonable, predicting system complexity and reallocating functions that have been deemed too complex saves time and money and produces a system with fewer complex components.

Ideally, system and software complexity metrics should be used throughout the development life cycle to continually assess complexity. Using metrics to determine and reduce complexity would contribute to the production of systems that are easy to understand and analyze. However, the metrics that have been defined for systems and software are directed primarily at implementation cost and maintenance effort estimation. It is not obvious that they can be extended to address performance issues.

2.1 System Complexity

As used in this report, the word “system” refers to the entire control or instrumentation system, including computer hardware, computer software, communications, instrumentation and human operators. The control system is only part of a larger system, which also includes the process being controlled, other humans and affected portions of the environment.

2.1.1 System Complexity Defined

System complexity is a measure of how difficult a system, composed of programs, computers, other hardware, natural or artificial processes, human beings, and the interconnections between all of them, is to understand and maintain. The problem of estimating system complexity is so difficult that little more than qualitative results exist. Most quantitative measures apply to software only.

System complexity metrics rely on the number of interfaces (human, software, and hardware), and the number of programs, modules, functions, and files within the system to determine complexity. Since

system complexity metrics are still experimental, system evaluators tend to use heuristic measures to evaluate systems.

System complexity can be influenced by a number of factors, including the number of components (e.g., hardware and software) in the system, the allocation of functions, the number and type of interfaces between components, and the number and sizes of files to be processed. Requirements for system safety, fault tolerance, real-time processing and operator involvement also affect complexity.

2.1.2 System Complexity Evaluation

System complexity is a major problem facing today's developer. Reducing complexity during system development saves time, money, and problems once the system is placed into production. System complexity depends upon the allocation of functions the system must perform. If a single function is allocated between multiple system components, the additional data passed between the function segments will increase complexity. In any system there is a point at which lowering the number of functions per component and increasing the number of components will increase system complexity. The higher number of components requires added data channels and data flows that will increase system complexity while keeping module complexity low. System complexity can therefore be reduced by grouping similar functions to minimize the amount of data passed between modules.

Database requirements affect system complexity. As the number and types of files to be processed increase, the complexity will increase. The requirement to process a relational database or provide multiple-user access to the database produces a complexity level higher than do the requirements for a single-user access flat database file.

Requirements for system safety and fault tolerance impose additional complexity through the processing necessary to respond to faults, and the circuitry or processors necessary to compensate for single-point failures. Real-time processing requirements may add complexity by adding to the number and speed of data channels to sense and respond to system conditions, and the number of processors needed to meet response time requirements. In control systems where the goal is to minimize operator involvement and administrative procedures, the additional processors to accomplish this goal will add complexity to the computer system. Data channels to provide sensor data to these processors also add complexity. One goal of system design is to balance this added complexity against the added functionality, safety and performance of the system.

2.1.3 Complexity Attributes

Due to the scarcity of system complexity metrics, the idea of system complexity attributes has been devised. This is a common-sense approach to measuring system complexity, but has not been proven. A less precise alternative to formal complexity metrics is to identify the use of practices that have added difficulty in the past usage, are difficult to analyze and verify, or that are still topics of research and controversy. The presence of these attributes then is presumed to indicate difficulty, risky techniques or algorithms, or controversial approaches. The British Ministry of Defence (MoD) takes this stance (MoD 1991a) when it rejects certain practices that cannot be analyzed by currently available formal methods. Other areas of potential difficulty are specifically mentioned in testing and performance analysis requirements (MoD 1991a,b). In the following discussion, a list of six complexity attributes (not exhaustive) is introduced and justifications are given for items in the list:

- Timing complexity.
- Priority complexity.
- Resource complexity.
- Connection complexity.

- Concurrency.
- Communication delay.

Some practices discouraged by MoD (1991a) are not included: floating point arithmetic and recursion, because they are algorithmic[1] instead of structural; interrupts, because interrupts are subsumed by priority and resource complexity. The concept of concurrency is used in a more restrictive sense than in MoD 1991a.

Timing complexity refers to the additional complexity incurred by scheduling sequences of events to occur at future times. In the simplest polling loop systems, a sequential loop of actions is repeated endlessly. In a more complex, scheduled system, actions may be scheduled to occur at future times unrelated to polling frequency. The introduction of future scheduling means that the system no longer responds in strict order of input, but can reorder output responses. Verification of timing is specifically required in MoD 1991a 35.5 (c), MoD 1991b 30.5.1.3, MoD 1991b 33.4.2 (c), and MoD 1991b 35.7 (c). Timing and time relations in real-time systems continue to be topics of current research (Fidge 1991). Modern real-time systems still exhibit timing errors (Riezenman 1991).

Priority complexity introduces an additional concern of precedence. If two or more actions are scheduled (or initiated by interrupt)[2] to occur at the same time, a priority scheme determines which action shall be executed first. Priority schemes introduce the additional complication of synchronization, and may reorder output responses both by time and precedence. Priority and scheduling are still hot topics in real-time system research (Sha et al 1990, Sha & Goodenough 1990).

When real-time systems become multi-tasking or multi-threaded,[3] more complexity is introduced when system resources are shared between tasks or code threads. With priority schemes and resource management, deadlocks become a potential problem. Deadlocks have been a source of trouble in many large systems, for example see Kleinrock 1978. MoD 1991b 30.5.1.3 specifically names deadlock as a subject to be investigated during performance analysis. The problem of scheduling and resource allocation in real-time systems is still a topic of research (Zhao 1987). Resource complexity is the object of one metric in the literature (Hall & Preiser 1984).

Connections in a large real-time system can be logical or physical. If the system is implemented as a number of tasks (Hoare 1978), the tasks may be connected by inter-task communications. These connections are logical and may be implemented as operating-system-mediated inter-process communications[4] in a single processor, or as some combination of logical and physical links if the tasks are running on a multiprocessor system. MoD 1991b 31.3.3 (c) requires that Static Path Analysis be performed and that software should be neither "over-modularized" nor "unduly monolithic." Connection complexity is an extension of these concerns to the complication of intertask communication and cooperation, where Static Path Analysis may be difficult or impossible.

While a single-processor, multi-tasking real-time system may exhibit apparent concurrency,[5] only multiprocessor systems can exhibit true concurrency, which is the kind under consideration here. MoD 1991a 30.1.3 prohibits concurrency in the wider sense of apparent concurrency. True concurrency is defined as two or more instruction streams being executed simultaneously, which requires two or more

---

1 Both can be replaced by other ways of doing the same thing.
2 MoD 1991a 30.1.3 prohibits interrupts which are difficult to analyze because of unplanned preemptions, a simple form of prioritization.
3 MoD 1991a 30.1.3 prohibits concurrency, in this case apparent concurrency.
4 See Preckshot and Butner 1987 for a description of such a system. The inter-process communication system (IPCS) may extend over multiple processors and may be operating-system independent.
5 Brinch Hansen 1975, for example, defines concurrent programming constructs which are actually executed serially on a single processor. Apparent concurrency is concerned with synchronization and code segments called critical sections.

instruction execution units (essentially, two or more CPUs). With true concurrency, it is no longer possible to assign strict time ordering to instruction streams in the real-time system, with concomitant increases in complexity in the concepts of simultaneity and synchronization. These difficulties make true concurrency, or timing in distributed systems, a current research topic (Fidge 1991).

Communication delay not only introduces additional uncertainty into intermachine timing, but significantly complicates recovery from failures. Since some delay is normal and expected, failure recovery must include waiting periods to ensure that recovery procedures are not started when there actually is no problem. MoD 1991b 30.5.1.3 (i) addresses the issue of communication delay, but not in the more difficult context of fault recovery. Current research (Muppala et al 1991) is directed toward system response and deadline performance in the presence of faults.

The design of a large real-time system involves a great many different factors. System complexity measurement is one method used to determine the proper balance among the factors. For example, a layered distributed approach to system design can achieve functional isolation, leading to improved safety and performance. The taxonomy given in the next section for system complexity should be viewed as descriptive. While a complex system is generally more difficult to analyze and test than a simple system, the complexity may be necessary to achieve important system goals. One must look at the entire problem being solved when evaluating the complexity of the system.

2.1.4 Simplified Taxonomy of Real-Time Systems

Using the complexity attributes just presented, real-time systems can be classified into groups, based upon presence or absence of attributes. Since such a classification scale is not validated against some software quality, it should be considered as being plausible only. The taxonomy proposed below is similar to the classification scheme used in biology, wherein products of later evolution are generally larger and more complex. Like biological evolutionary taxonomies, the ordering in this taxonomy may be imperfect. To validate this scheme as described in Rombach 1990, a target quality (goal) would first need to be selected—for example, the effort needed to devise, use, and analyze a performance model.[6] This effort should be accurately quantified (in man-hours, for instance) and then measured for a reasonably large number of candidate real-time systems. This provides a sample ensemble consisting of ordered pairs, the real-time system taxonomy level, and the effort required to estimate its performance. The ensemble is examined for correlation between level and effort. The hypothesis that taxonomy level is a predictor of performance estimation effort is conditionally sustained if the correlation is high enough, and falsified if there is no correlation.

An assumption is made in this taxonomy that real-time systems contain loops, and may optionally deal with preemptive events. This is true in the broad sense that any finite discrete system eventually repeats itself, but is also true in that real-time systems are often cast as repetitive tasks. Using the attributes sidesteps the issue of code size. The results of Henry and Kafura (1981) in particular, and the general uncertainty of Source Lines of Code (SLOC) or size as a complexity measure, seem to indicate that structural measures are more reliable indicators of complexity. Relying upon software structure alone for classification ignores a number of knotty practical problems such as visibility into system internals and the effects of hardware on software operation. In the following taxonomy, hardware is considered in a very broad sense,[7] and operating systems are not explicitly separated from other code. Where operating

---

[6] Devising the performance model may be the simplest task. Deciding what simulated loads to apply and interpreting the results may consume a large part of the effort.

[7] For far more detail on hardware, see, for instance, Savitzky 1985 chapter 2.

systems are used,[8] the system should be classified as the more complex of either the operating system, the real-time application code, or the combination of the two.

2.1.4.1 Single-Processor Systems

In most cases it is simple to decide the number of processors in a system. However, coprocessors and auxiliary processors such as vector accelerators add confusion. Systems with closely coupled coprocessors such as math coprocessors are considered to be single CPU systems. Smart interfaces, such as Ethernet interfaces, Small Computer System Interface (SCSI) controllers,[9] and some video accelerators still do not constitute a multiprocessor system. However, a separately programmed auxiliary processor which executes almost-general-purpose code in parallel for significant time periods probably qualifies as an additional CPU.

2.1.4.1.1 Polling Loop (Level 0)

The simplest of all real-time system organizations is the polling loop. This system exhibits no complexity attributes, and is a single loop with sequential execution and (almost) no interrupts. Branches are allowed to select alternative actions depending upon input values. A single timer interrupt is sometimes allowed as long as it is used only to update a clock or time variable. The polling loop executes continuously and endlessly, and the loop time is determined only by the amount of work performed during each iteration.

In terms of total number of systems extant, polling loops and scheduled polling loops may be the most popular types of real-time systems by a wide margin. Simple Programmable Logic Controllers (PLCs), embedded digital controllers, and simple, ad hoc laboratory control systems all benefit from the simplicity (and hence reliability) of the polling loop.

In spite of the simplicity of the polling loop, it is still possible to ruin its reliability through poor design. Poorly designed input/output interfaces can stall the loop by causing it to wait for something that never happens.

The scheduled polling loop is still a level 0 organization. Simple polling loops run at whatever speed the loop load allows. Some applications, such as digital signal processing, digital filters, proportional-integral-derivative (PID) controllers, and other applications of digital techniques to continuous-time systems, require precisely timed input/output cycles. The polling loop can be extended to cover these applications by starting each loop cycle at precisely timed intervals. The scheduled polling loop does not exhibit the timing complexity attribute because outputs still occur in the order of inputs. In order to perform correctly, loop time must not exceed the precise timing interval, but this can be determined by static timing analysis.

2.1.4.1.2 Schedule Loop (Level 1)

Unlike the polling loop, the schedule loop does not visit a fixed list of inputs and tasks,[10] but instead continually polls a dynamic schedule. Things-to-do are placed on the schedule either repetitively, by themselves, or by other, previously executed things-to-do. Inputs can be polled at scheduled times, so the schedule loop reduces to the scheduled polling loop as a degenerate form. The schedule loop exhibits the timing complexity marker, but in this instance timing correctness may only be verifiable dynamically, unless strict rules are followed for self-rescheduling or cross-scheduling. Schedule loops can be used for a surprising number of real-time applications, including flight control systems and mission supervisory control systems on spacecraft.

---

8 We consider run-time kernels in embedded systems to be operating systems.
9 See Savitzky 1985, chapter 2.
10 Here task is defined as something to do, as opposed to the definition found in the glossary.

2.1.4.1.3 Preemptive Single Task (Level 2)

Both timing and priority complexity attributes are exhibited by preemptive single task real-time systems (often called "event-driven systems"). Any kind of preemption introduces a priority scheme, whether explicit or implicit. Preemption introduces unplanned delays in the preempted program in order to service higher-priority demands with minimal delay. The typical preemptive real-time system services interrupts while running a real-time application (such as a schedule loop) as the non-interrupt single task. Since the application may be in a delicate state at the time an interrupt occurs, these systems often have methods to synchronize the incoming data from the interrupt with an opportune point in the application code.

Although not formally called tasks, interrupt service routines actually execute in different contexts from the real-time application, and thus fulfill at least some of the requirements for being called tasks. When completion routines are executed as a result of interrupts, the distinction blurs further, since completion routines,[11] signal handlers,[12] and asynchronous trap routines[13] are usually considered a form of light-weight task. Nonetheless, we consider systems with such features as single-task systems, mostly because the issue of resource management is not raised. In single-task systems, all resources are available to the real-time application task, and there is no resource contention.

2.1.4.1.4 Preemptive Multitasking (Level 3)

Timing, priority, and resource complexity attributes are displayed by preemptive multitasking single-processor real-time systems. If the tasks communicate, logical connection complexity is present as well. The multitasking preemptive system is similar to the single-task system, but now several tasks (which appear to be separate code threads in different contexts) may contend for execution time and for system resources. This level of complexity is the first level at which it is common to see subtle errors due to interactions between timing, priority, and resource allocation. Deadlock is the most feared problem, but performance deficits due to resource contention may be more common and harder to detect.

Foreground/background systems are common examples of small, preemptive multitasking single-user systems,[14] but almost any combination can be found in embedded, non-interactive real-time systems.

2.1.4.2 Multiprocessor Systems

When a single processor can no longer support the required computational load, or when data and input/output devices are physically separated, additional CPUs are added to increase total processing speed or to manage data and devices in local areas. Many ways exist to connect CPUs together; they may be roughly divided into closely coupled versus loosely coupled systems. The principle for separating systems is that closely coupled systems communicate between CPUs almost as fast as the CPUs can process, thereby making it practical to subdivide processing jobs rather finely and still retain an advantage. Closely coupled systems tend to be coupled by shared memory mechanisms, and loosely coupled systems usually are connected by serial lines or networks such as Ethernet or fiber ring. There are exceptions to this,[15] just as there are exceptions to the generalization that closely coupled systems are usually used to increase processing power, while loosely coupled systems are used to manage physical

---

11 Digital Equipment Corporation (DEC) RT-11 operating system synchronization mechanism.

12 UNIX operating system synchronization mechanism. Its mention here is not meant to imply that UNIX is a real-time operating system.

13 DEC VMS operating system synchronization mechanism.

14 Usually a small computer system which performs real-time chores as the foreground task while attending to its master's voice in the background.

15 For example, Transputers are interconnected by very fast serial links, not shared memory. Transputer systems are considered closely coupled, and are commonly used to increase total available processing speed.

separation. Further discussion of specific examples of networks and data communications is left to another report.

To be considered a multiprocessor system, code running on one processor must be dependent upon code running on another. Some combination of resource or connection complexity must tie concurrently executing code together (see Lehman and Belady 1985 for a discussion of what constitutes a system), or the software must exhibit interdependencies through the physical system being controlled[16] (e.g., actions by one software module cause physical system changes upon which another software module depends for proper operation). Independent real-time systems that do not affect each other should be considered at their individual complexity attribute levels.

2.1.4.2.1 Closely Coupled Systems (Level 4)

Closely coupled systems are usually in the same room or in the same electronics rack, and are usually connected by wide (many conductor) memory buses.[17] A useful discussion with examples of actual commercial shared-memory or shared-bus multicomputer systems can be found in Weitzman 1980, chapters 2 and 4. Because several connected processors may attempt simultaneous access to shared memory, there must be at least simple hardware arbitration to prevent inconsistent data from being written to shared memory. Often there is very little more than this, and software bears the burden for higher-level coordination. However, some processors, such as the INTEL 432 and the INTEL 960, have substantial functions built into their hardware to support system-level shared memory management, and also to support shared-memory, multiprocessor operating systems.

Symmetric Systems. Less complex among closely coupled systems are symmetric multiprocessor systems, where both hardware and software[18] are replicated as more processors are added. These systems exhibit at least timing, connection, and concurrency complexity, and are likely to exhibit priority and resource complexity as well. Performance models may have to include models for processor scheduling. In systems with replicated tasks but different data, performance models are simplified, since the model can be replicated as well.

Asymmetric Systems. If a system is asymmetric, some hardware and software differ depending upon which CPU is under consideration, which increases the complexity of the performance model. These systems exhibit at least timing, connection, and concurrency complexity, and are likely to exhibit priority and resource complexity as well.

2.1.4.2.2 Loosely Coupled Systems (Level 5)

The component parts of loosely coupled systems are often separated by hundreds of meters, and are usually connected by some sort of network technology. Communication delay is a significant factor, and is included *ab initio* in the design process. These systems exhibit at least timing, connection, concurrency, and communication delay complexity attributes, and are likely to exhibit priority and resource attributes as well. While it is possible to have a single "network" operating system, typical loosely coupled multiprocessors have separate operating systems running on each CPU.

Symmetric Systems. These systems have identical nodes emplaced in a symmetric communication network. The software running on each node is the same, but the data or sensors may be different, requiring communication to obtain non-local data. Performance modeling is simplified because each node is replicated and embedded in the performance model for network communications.

---

16 This is an unfortunate situation, because software correctness becomes dependent upon correctly understanding details of the physical system.

17 Except, of course, Transputers.

18 There may, in fact, be a single distributed operating system and a set of distributed application tasks which run on whatever processors are ready.

Asymmetric Systems. If a system is asymmetric, then some hardware and software are different, depending upon which CPU is under consideration. This increases the complexity of the performance model, which may include different models for each node and an asymmetric communications model.

2.1.4.2.3 Hybrid (Level 6)

Hybrid systems consist of processors of various design connected by networks in clusters of shared-memory multiprocessors. Such systems have been used successfully in large control systems (Wyman et al 1983). These systems are the most complex that one can encounter, and usually exhibit all complexity attributes.

2.2 Software Complexity

2.2.1 Software Complexity Defined

In contrast to system complexity, software complexity is an indicator of how difficult a program, module, or the functions performed are to understand or modify. It is generally agreed that a software module must be at least as complex as the function it is supposed to provide. Quite often, software is more complex than this minimum, and one objective of measuring software complexity is to estimate both the minimal and actual complexity. Historically, producers of software have had the most motivation to attempt measurements of complexity and other software attributes that have economic impact.

The science of measuring software is called "software metrics." Metrics exist for size, cost, style, difficulty, structure, errors, and reliability, to name several. Aspects of software complexity have been discussed by several authors in widely varying ways, because the concept of complexity is inherently disordered. There is probably no finite set of numbers which can describe even a partial ordering from simple to complex. The current practice in devising software metrics, nicely summarized by Rombach in the following paraphrase, provides both an approach to selecting aspects of complexity to measure and a framework for understanding metric research.

0. Define a goal.[19]
1. Model the quality of interest and quantify it into direct measures.
2. Model the product complexity in a way that lets you identify all the aspects that may affect the quality of interest.
3. Explicitly state your hypothesis about the effect of product complexity on the quality of interest.
4. Plan and perform an appropriate experiment or case study, including the collection and validation of the prescribed data.
5. Analyze the data and validate the hypothesis.
6. Assess the just-completed experimental validation and, if necessary, prepare for future experimental validations by refining the quality and product-complexity models, your hypotheses, the experiment itself, and the procedures used for data collection, validation, and analysis.

(Rombach 1990)

[19] Step 0 was abstracted from Rombach's goal/question/metric (GQM) paradigm description as being a reasonable preamble to the steps described for the author's Distos/Incas experiment.

Defining a goal may seem an obvious precursor to devising a metric, but it may not be an obvious precursor to choosing a metric from a catalog of existing metrics. Metrics are validated by experimentally determining the correlation between the metric value and the particular aspect or quality of software which the metric presumably affects. Metrics in the literature have been validated against such software qualities as development effort, debug time, errors per line of code, difficulty, or changes per software module during maintenance. These qualities often have narrow scope and it may be difficult to conclude that they are related to other goals and purposes.

A software developer may wish to measure certain types of complexity in a design-in-progress for the purpose of moving effort or scrutiny to where it is most needed. A software tester may use metrics to determine where to direct the most testing effort. The NRC may wish to require certain maximum levels of complexity for safety-critical software. For this it would need a measure or attributes to describe unacceptable complexity. Reviewing a system design after it is complete, the NRC will be concerned with estimating the effort required to validate a design, estimating the effort required to predict system performance, or estimating the hazard level of a design. In these cases, metrics would be needed that were correlated with validation effort, the effort to implement and use a performance model, or the number of hazardous errors contained in either the design or its implementation. As a practical matter, any metric chosen or devised would also have to be calculable with the data available at the time it was needed.

2.2.2 Software Complexity Metrics

A large number of complexity metrics for software have been invented. Most of these are aimed at reducing the cost of developing or maintaining software, or at increasing its correctness. It is not obvious that measures relating to such aspects can be directly extended to issues relating to performance, reliability or safety. Thirteen of the more popular complexity metrics are described on the next few pages.

2.2.2.1 Source Lines of Code (SLOC) Metric

One of the metrics in software development has been delivered SLOC; the intent has been to measure effort required to complete a software project. This metric was chosen mostly because it was easy to determine and because it seemed reasonable that effort was proportional to the amount of code written. Source lines of code has had indifferent success either in predicting software effort or in recording expended effort (Brooks 1975). "Lehman and Belady (1985), in attempting to characterize large systems, reject the simple and perhaps intuitive notion that largeness is simply proportional to the number of instructions, lines of code, or modules comprising a program" (Burns & Wellings 1990). In another experiment, debugging effort was shown to be independent of SLOC (Gorla et al 1990). One reason given for imprecision in SLOC is that some programmers are prolix while others are terse (Boehm et al 1984). Regardless, SLOC continues to be used either because it is easy to compute or because it is used as a yardstick against which other metrics are compared.

2.2.2.2 McCabe's Complexity Metrics

Because of the poor success of SLOC or size metrics at quantifying difficulty or expected effort, other metrics more related to debugging, implementation, or maintenance effort have been proposed. Rather than code size, these metrics attempt to measure some aspect of software complexity and use it to predict or quantify the target software quality of interest. McCabe (1976) suggested a graph-theoretic metric which measures complexity in terms of linearly independent paths through the code. He calls this "cyclomatic" complexity. A second complexity measure, which he calls "essential" complexity, measures how much the complexity of a program module can be reduced by conversion of single-entry/single-exit (SE/SE) subgraphs to vertices (i.e., subroutine calls). Essential complexity is the cyclomatic complexity

with all SE/SE subgraphs converted to subroutine calls. If the essential and cyclomatic complexity are equal, the module's complexity cannot be reduced. If essential complexity is less, then the cyclomatic complexity can be reduced. McCabe suggests that a complexity greater than 10 (McCabe & Butler 1989) is difficult for programmers to comprehend, and will therefore be difficult to develop and maintain. Part of the motivation for this metric was to improve the testability of routines by putting an arbitrary limit on size. This metric requires access to code or to a graphical representation of the module design,[20] and is useful at the module or subroutine level. McCabe presents examples of use and gives anecdotal reports of metric usage. There is no formal validation, but others (to be described) have used McCabe's metric in validation experiments. McCabe's metrics are being used by some software developers to help reduce control flow complexity.

2.2.2.3 Software Science Metrics

Halstead (1977) introduced "software science" metrics based upon counting operator and operand tokens in source code. Various arithmetic combinations of operator, unique operator, operand, and unique operand counts are used as measures of length (N), volume (V), level (L),[21] difficulty (D),[22] and effort (E).[23] Basili et al (1983) examined the correlations between SLOC, Halstead's, and McCabe's metrics with coding effort and number of errors for seven projects and 1,794 Fortran modules at NASA/Goddard Space Flight Center. This represents an independent attempt to validate these metrics against two software qualities: coding effort and number of errors. None of the metrics tested consistently showed more than moderate correlations with effort or errors. The Basili paper concludes:

1. None of the metrics examined seem to manifest a satisfactory explanation of effort spent developing software or the errors incurred during that process.
2. Neither Software Science's E metric, cyclomatic complexity nor source lines of code relates convincingly better with effort than the others.
3. The strongest effort correlations are derived when modules obtained from individual programmers or certain validated projects are considered.
4. The majority of the effort correlations increase with the more reliable data.
5. The number of revisions appears to correlate with development errors better than either Software Science's B metric, E metric, cyclomatic complexity or source lines of code.
6. Although some of the Software Science metrics have size-dependent properties with their estimators, the metric family seems to possess reasonable internal consistency.

(Basili et al 1983)

2.2.2.4 Information Flow Metric

More convincing correlations were reported by Henry and Kafura (1981) using a metric based on "information flow." This metric measures complexity in terms of data connections "fanning in" and "fanning out" from modules. The Henry metric represents a significantly different approach from the Halstead–McCabe metrics. It is a "structure" metric, as opposed to a "code" metric (Henry and Selig 1990). As a validation for the metric, Henry used the UNIX operating system[24] as the study subject, and

---

20 See Henry and Selig 1990 for an example of applying this metric prior to coding.
21 The theoretical minimum number of bits required to code divided by the actual bits required to code.
22 1/L, supposedly the difficulty required to code the algorithm.
23 V/L, supposedly the effort required to understand the implementation of the algorithm.
24 "The third reason for selecting UNIX is the fact that UNIX software was designed for users and not as a toy or experimental system. In particular, it was not written to illustrate or defend a favored design methodology" (Henry and Kafura 1981).

the number of changes per module during life cycle maintenance phase as the quality to be predicted. Henry reports an impressive 0.98 correlation between percent-of-modules-changed versus a metric defined as:

$$(\text{fan-in} \times \text{fan-out})^2$$

Notably, she achieved better correlations when a size-dependent term was removed from the expression.

In a later experiment, Henry examined the performance of the information flow metric, SLOC, cyclomatic complexity, and the Halstead Software Science metrics on programs written by senior-level software engineering students at Virginia Polytechnic Institute and the University of Wisconsin at LaCrosse (Henry and Selig 1990). Using students as sources of research material is controversial because, as critics point out, the academic environment differs markedly from the commercial and professional environment. In particular, academic projects are often small, simple, and relatively well constrained as far as what problems can be addressed. In this case, Henry reports that the projects were large (3000–8000 SLOC), the specifications were not provided by the professor, and the students had no access to experimental statistical data. The metrics were applied in three design refinement levels to 276, 283, and 422 software modules respectively. The numbers and distribution of student programmers involved were not mentioned. The results show a wide range (a factor of two) between predicted and actual complexity for the information flow metric at 95% confidence level.

2.2.2.5 Style Metric

An example of a "style" metric is provided by Gorla et al (1990) in an attempt to predict debugging effort from style attributes (lexical complexity) in Cobol programs. These attributes include such things as percent comment lines, percent blank lines, number of goto statements, indentation, and the like. As controls, SLOC and cyclomatic complexity were included. The validation attempt used 311 student programs collected from five intermediate Cobol programming classes as raw material. Gorla reported that nine of fourteen metrics had values which correlated with minimal debug time (e.g., variable names between 12 and 16 characters long correlated with short debug times). SLOC and cyclomatic complexity showed no significant correlation with any of three debug measures.

2.2.2.6 Information Theory

Sahal (1976) described an information-theory-based system complexity metric which measured complexity based on four behavioral components, viz. organized, unorganized, short-term, and long-term behavior. These roughly correspond to deterministic, non-deterministic, short-term time series, and long-term patterns of behavior. Sahal presented an application of his model to the evolution of the technology of piston-engined, carrier-based aircraft. The sample size is one, which makes the interpretation of the claimed correlations unconvincing. The idea of a system complexity metric with a mathematical basis is attractive, but a significant amount of research would be required to apply this metric to software systems.

2.2.2.7 Design Complexity

Working from software structure charts, McCabe and Butler (1989) extended their module cyclomatic complexity metric to graphs that describe how code modules work together, rather than how individual modules work. "Module design complexity" is a per-module complexity measure that describes the number of unique basis paths by which a module decides to call subordinate modules. "Design complexity" is the sum of the individual module design complexities taken over the entire software structure chart. "Integration complexity," the third metric of this group, is a measure of the integration

tests required to qualify the design. Like the original McCabe cyclomatic complexity metric described above, no validation results are presented in McCabe and Butler 1989.

2.2.2.8 Graph-Theoretic Complexity

Another attempt to apply McCabe's cyclomatic complexity measure to networks of modules resulted in the "graph-theoretic complexity for architecture" metrics (Hall and Preiser 1984, IEEE 1988b). The intent of this approach was to devise complexity metrics that accommodated concurrent systems modeled as networks of potentially concurrently executing tasks. These complexity metrics require system representation as a strongly connected graph, with the nodes being software modules (tasks) and the edges being transfers of control (messages). "Static complexity" is then defined as:

$$C = \text{edges} - \text{nodes} + 1.$$

"Generalized static complexity" combines a complexity estimate for program invocation with resource allocation complexity, summed over the elements of the network graph. This metric attempts to measure the complexity associated with resource management, synchronization, and communication. The third metric, "dynamic complexity," is static complexity evaluated for actual module invocations over periods of time. It is intended to account for the effects of actual frequency of invocation and module interruptions. Published reports of usage of these three metrics in the open literature are limited. Generalized static complexity and dynamic complexity have not been reported in operational use (IEEE 1988b).

2.2.2.9 Design Structure Metric

The "design structure" metric (IEEE 1988a,b) produces a complexity number that is the weighted sum of six "derivatives" calculated from seven measurement primitives. The measurement primitives are counts of modules or database elements having certain qualities, rather than internal details of the modules themselves. The metric attempts to quantify dependence on input/output, dependence on stored state, database size, database compartmentalization, and excess module entries/exits. Reported experience for this metric is limited and not easily accessible.

2.2.2.10 Function Points Metric

Albrecht and Gaffney (1983) present a "function points" metric that uses external system stimuli and functional requirements as its inputs. As an alternative to direct size estimation, function points attempt to characterize software projects by input/output connections and the difficulty of functions performed. The metric is therefore applicable at an earlier stage of the design process, when little or no coding has been done. Albrecht reports correlations over 0.9 with "work-hours," a measure of time required to implement an application. However, relative standard deviations for Albrecht's metrics are in the range 0.4 to 0.8, which suggests a typical relative error of a factor of two. Function points are computed using "external input types," "external output types," "logical internal file types," "external interface file types," and "external inquiry types." These items appear to be closely related to business data processing practices, as do the detailed rules for counting and combining them.

2.2.2.11 COCOMO

The COnstructive COst MOdel (COCOMO) model (Boehm 1981) for estimating software project effort and expected project time is an empirical model relating SLOC with expected man months and total months required to complete a project. The basic COCOMO model uses two simple equations to compute man months and project time:

$$MM = K\,(SLOC)^{P}$$

$$TIME = R\,(MM)^{q}$$

K, p, q, and R are functions of project team type. How the original estimate of SLOC is arrived at is never explained, but in early project stages SLOC estimates are notoriously bad, and the exponent p is greater than one. The accuracy of the basic model is claimed to be worse than a factor of two about 40% of the time, based on validation efforts on a database of 63 projects. The detailed COCOMO model is a weighted-SLOC model with 15 cost-driver weights. This is a complex model requiring a lot of subjective judgments. The accuracy claimed for the detailed model is no more than 20% error from actual development effort 58% of the time.

2.2.2.12 Cohesion

The cohesion metric (Yourdon and Constantine 1979) is an indication of how well a module conforms to a single function. Single-function modules are usually less complex and more understandable than larger multifunction modules. Modules with strong cohesion perform only one function. When design changes are needed or errors occur, these modules can easily be understood and changed. A module that performs a group of related functions in a logical order is more complex than modules that perform one function. A randomly cohesive module performs related functions in a random order. Using the same module to update several databases and retrieve data from them is an example of random cohesion. Modules with weak cohesion process unrelated functions in any order and should be avoided.

2.2.2.13 Coupling

Coupling (Yourdon and Constantine 1979) describes how well a module interfaces with its units and how well the units interface with each other. The coupling metric measures how well modules and units pass data. The best type of coupling is data coupling; only the required data is passed from unit to unit. Stamp-coupled units pass entire data structures instead of just the necessary data. Externally coupled units share the same global data items, but not the entire global data base. When two or more units share the common database they are common-coupled. Common coupling allows any unit to access and modify common or global data, adding to complexity by expanding the number of places that data can be accessed. Lastly, content coupling means that one unit can directly reference data inside another unit. Content coupling is the least desirable form of coupling.

3. Scalability

An important question is, “At what point does a small, fast, real-time system, when extended to the large, become a big, slow failure?” The property that describes the ease of extension of small to large is termed “scalability.” Scalability includes both the extension of actual small systems to large systems and the application of small system techniques inappropriately to the large system domain (thinking small). Unfortunately, knowing how to scale real-time systems is more a matter of experience than theory.

“Only when we have built a similar system before is it easy to determine the requirements in advance” (Parnas 1985).

Most of what is known has been gained by experience, not forethought, because scaling difficulties tend to be specification oversights, not software development method errors. Some techniques have been described for anticipating system scaling (Parnas 1979) but retrofit to existing systems is problematic. Some issues in system scaling, such as capacity and resource usage, are susceptible to quantitative

modeling, and performance is usually predictable, provided that realistic load patterns are used to drive the models. Other areas are more difficult to predict and may produce unpleasant surprises.

3.1 Synchronization

The first and most obvious effect of replicating CPUs and connecting them together is that time is no longer the same at different points within the system. At the very least, complete time ordering degenerates to partial orderings of events occurring within each CPU. Current research in real-time systems focuses on how to ensure that supersets of such partial orderings in real-time distributed systems are causal (Fidge 1991). The state of practice is not reassuring, and synchronization and time ordering within distributed real-time systems is an area of continuing research. Real-time system developers continue to make timing and synchronization errors.

Timing errors show up as incorrect (or mismatched) times associated with events or objects, or as incorrect ordering of events, data, or messages. The range of errors this can cause is enormous. The system can drop data because of apparently incorrect time stamps, behave erratically because of incorrect order, deadlock waiting for something out of order, or freeze completely because fault detection circuitry or software stops the system on synchronization failure.

Arguably the most famous public example of a system freeze may be the Space Shuttle synchronization error (between four redundant CPUs) that caused a delay during the first Shuttle launch attempt (Lombardo and Mokhoff 1981). This was caused by the elementary error of using different clocks for different processes, resulting in a time skew between processes. This would not have been detected (and may not have been significant) in a single processor system, but was crucial in a cross-checking multiprocessor system. Fortunately, this was resolved with no injuries except to the pride of IBM Federal Systems Division. That it could happen to such a large and experienced organization is indicative of the difficulty of synchronizing distributed systems.

3.2 Precision

Precision refers to the ability of a variable to represent some quantity important to the functioning of a system. Precision limitations can exacerbate the effects of poor design, or they can engender sudden, unexpected difficulty when enlarging some system dimension, typically an address or name space.

The designers of the PDP-11[25] have reported that they consider their worst design error to have been limiting the address representation of the machine to 16 bits, which makes the maximum direct address 65,536 (64 Kbytes). Later, Intel repeated the error, more creatively, in their 80x86 architecture microprocessors, and Microsoft enshrined it in the MS-DOS operating system, which, even after five major versions, is limited by the well-known "640K barrier." The architectural limitation is so pervasive that many compilers for 80x86 computers do not support data structures larger than 64 Kbytes.

The most recent public failure due to time precision was the failure of some Patriot missiles in the Gulf War. Initial reports cited an "imprecise timing calculation" that resulted in a cumulative error of 360 milliseconds after 100 hours of Patriot battery operation (Riezenman 1991), which caused the system to drop an incoming Scud missile from its target list. A later GAO report more accurately stated that the error resulted from an integer-to-floating-point conversion that had only 24 bits of precision. Since the clock hardware had integer registers, the error grew proportionally to the time the system had been

[25] A Digital Equipment Corporation computer circa 1972.

running without a restart. This was a system specification error caused by a lamentable lack of foresight on the part of the system specifiers, who failed to anticipate system operation expanded along the time axis. Real human operators sometimes forget to compensate for built-in system deficiencies, as was the case in the Gulf.

Precision limitations may show up as a frank failure to adapt to larger scale applications, but unfortunately the more usual symptoms are progressively more contorted attempts to "band-aid" the system into operation, with decreasing reliability and performance. The effects are insidious and often are not acknowledged until a failure or severe performance bottleneck appears under load or fault recovery.

3.3 Capacity

Systems sometimes run into severe performance problems when the capacity of an innocuous component proves to be the system bottleneck. The OS-32 operating system[26] provides a classic example. This system did not spool operator error messages, but instead held the messages and the tasks sending the messages hostage to the speed of the operator's console. Naturally, when serious problems occurred, many tasks wrote messages to the operator's console, and the "real-time" system slowed to the speed of a mechanical teletype machine. The modern-day equivalent is the overloaded logger.

Network links are a significant source of capacity problems. Often, when a distributed system is expanded, the traffic between parts of the system is imperfectly understood, especially under fault conditions. This is exacerbated by the tendency of many to assume that raw network data rate is available to the tasks running in computers attached to the network. In fact, due to protocol overhead and operating system delays, the usual effective data rate for a computer attached to an unloaded Ethernet (1.25 Mbyte raw data rate) is 50–600 Kbytes. This ratio is typical of many networks.

Capacity problems occur because of insufficient analysis of data rates within the system and their effects on system devices and datalinks. When the system is small, potential capacity problems do not surface, because in absolute terms the data load does not stress the system, regardless of architecture. As the system is enlarged, more physical dataflows are then overlaid on datalinks and devices. The difference between the logical view and its physical implementation obscures pertinent detail. However, with the exception of certain effects discussed under the next heading, capacity expansion can be dealt with quantitatively by using performance models to predict the effects of increased load or different load patterns.

3.4 Load-Correlated Load

Linear estimates of capacity scaling can be vitiated by load derivatives that are positive functions of load itself. One area where this is likely to occur is in fault recovery protocols for transient, operationally expected faults. This can be particularly difficult to detect before the system is developed, because detection may require detailed analysis of fault handling algorithms and their interactions with architecture and computing environment. There also may be no sign of impending failure until a system actually becomes large and loaded. In small systems the rate of fault occurrence may be insufficient to incur much additional load during infrequent fault recovery. Expanding the system, however, may increase both the frequency of fault occurrence and the time required to recover from each fault. Now fault-handling protocols that are triggered during load-correlated system activities may cause sudden

[26] A multitasking operating system on Perkin-Elmer computers circa 1983.

system collapse above critical load levels. In effect, efforts to handle and recover from faults place additional load on the system, which causes further attempts to handle faults. At some load level, this is regenerative and system throughput drops almost to zero.

Early network communications research provides an illustrative example. Packet-switched carrier sense multiple-access (CSMA) data links have an unstable[27] throughput versus offered-load curve (Kleinrock 1975). This is because the error recovery procedure (simple retransmission) can multiply network load[28] at high packet collision frequencies. The Ethernet (Metcalfe and Boggs 1976) is stable under high load because the collision recovery procedures defer load rather than increase it (Shoch 1980).

The effect of human operators as impromptu fault recovery mechanisms should not be neglected. Operator actions that are intended to gain more information may place more load on the system, and operator impatience correlates well with slow system response, which in turn correlates highly with system load.

3.5 Intermediate Resource Contention[29]

Intermediate resources are resources required because an intermediary, often hidden, is placed in a logical dataflow when a system is expanded. They include buffers to store and forward data packets, disk buffers for database access, and locks, semaphores, and other operating system resources. At the worst, contention can result in deadlock (see Kleinrock 1978 for data communications examples), but, more insidiously, can result in performance degradation. While the previous comments on capacity can also be construed as a resource management problem, it is possible to have datalinks of adequate capacity whose performance is reduced by insufficient buffer allocation in one or more nodes. The condition, of course, only shows up during heavily loaded operation.

Resource problems occur because of insufficient analysis of resource usage within the system and, in extreme cases, because of total ignorance of resource usage. When the system is small, resource problems do not surface because very few resources are used. As the system is enlarged and more intermediaries are introduced, hidden resource choke points may be introduced as well. The problem is similar to the problem discussed under capacity, and avoidance techniques are much the same. Resource models can be included in capacity models, and quantitative estimates of resource usage can be obtained for different load patterns.

3.6 Databases

Whether explicitly recognized or not, each distributed system has an associated database. At the minimum, this database consists of embedded assumptions and data that are compiled into or entered into software that is running on nodes of the system. At the other end of the spectrum, data may reside in a formally defined database that is controlled by a distributed database manager. The minimal approach suffers from configuration management problems; it may be impossible to verify just what data is distributed throughout the system, and changing the system may be a nightmare. The formal database approach is flexible, but sometimes at the expense of performance.

---

[27] As offered load increases, throughput peaks and then drops.

[28] The packet is transmitted to completion during the collision, and then transmitted again during retransmission, at least doubling the effort required to send the packet.

[29] For more discussion of resources in real-time systems see Burns and Wellings 1990, Chapter 11. The general subject of resource management is more extensive than can be covered here.

An important class of systems, industrial process control systems, has at least two databases for each example of the class. One consists of associations between signal or point names and the actual sensors which produce data. Conversion factors and procedures may also be associated with the names, and the form of this data may be PLC programs distributed in various PLCs that make up the periphery of the system. The other database consists of operator displays and associations of signal names with locations in the display graphics. The two databases may be merged into one, depending upon implementation, and there probably are additional databases not mentioned here.

Database managers present logical views of data, regardless of the actual physical storage layout of the data. Because of this, database access time is a function of database size and the number of probes required to resolve indirections and to access physically separate locations. In a distributed system, a single query may generate a flurry of distributed queries as the datum or its components are located and retrieved. This can multiply the use of intermediate resources and datalink capacity to the point that, in large systems, the performance deficit may be fatal.

3.7 Correlated Stimuli

For economic reasons, in some distributed systems interconnections are undersized relative to the number of sensors available and amount of data potentially producible. To avoid overload, data is filtered at the source and only "meaningful" or "important" data is actually transmitted. This practice is justified and datalinks are sized by assuming that meaningful data can be described by a Markovian probability model and arrive at the boundary of the real-time system with exponentially distributed interarrival times. This assumption is made in the literature (see Kleinrock on queuing theory, for example) because it is mathematically tractable. Often, nothing could be further from the truth.

A real-time system undergoing an excursion often experiences large numbers of rapidly changing process variables that produce a large number of data correlated in time with the excursion. In addition, increased operator activity is directly correlated with lack of system response, which occurs when the system is overloaded. In systems which have variable data rates and which are undersized with respect to the maximum possible data rate, correlated data overloads should always be anticipated and the possibility carefully investigated.

4. Conclusions and Recommendations

Many factors must be combined to predict the safety, complexity and reliability of the system. The three major factors investigated in this report are system complexity, software complexity, and scalability. Software has a significant effect on the system performance because it is complex and difficult to verify.

4.1 System Complexity

No industry-accepted system complexity measure has been identified, and there is very little research in this area. A possible first alternative to firm system metrics is to make a preliminary assessment of system complexity using the complexity attributes presented in this report, or other, similar indicators. The preliminary assessment provides an idea of just how much trouble and effort to expect in determining system safety and performance. For safety-critical software, a complexity attribute level greater than one indicates possibly hazardous practices. For high-performance, real-time control systems,

a level greater than two indicates that there may be significant scheduling, resource, and coordination issues to be examined. A level greater than three indicates that there may be additional synchronization and performance issues which complicate testing and performance analysis. For any real-time system, a level of four or greater indicates that scaling issues should be closely considered. As remarked earlier in this report, complexity attributes have not been validated, and the previous advice must be considered to be based on an experienced estimate only.

4.2 Software Complexity

Complexity and quality metrics have been used to guide software development, predict modification difficulty, predict testing difficulty, estimate module modification rates during maintenance, and predict the number of remaining errors. The assumption underlying these efforts is that more complex modules are harder to understand, harder to modify successfully, and more error-prone. Before software implementation, this is taken to mean that high levels of measured complexity indicate that portions of the design should be re-engineered. After implementation it is taken to mean that re-design may be necessary, or higher maintenance effort and higher levels of remaining errors may be expected.

Attempts to validate the various complexity metrics against software qualities of interest have had variable results, with one important exception, fully supported by IEEE 1988b guidelines. That result is that the success of metric usage is extremely sensitive to measurement environment. Accurate software measurement is not the work of an afternoon or even of a few weeks, but must be embedded in a long-term program in which the metrics can be calibrated to the environment and to the people in the environment.

Another observation made about many of the metrics is that detailed application of the metric is tedious and error-prone. This is particularly true about code metrics, which require decomposition of source code into measurement "primitives." Many authors, as well as IEEE 1988b, recommend automation using source code scanner programs to reduce or eliminate tedium and errors.

The most effective way to use software complexity metrics is to begin using them before software design begins. Complexity metrics (and other metrics not covered in this report) should be a part of a software management program from the start. During design and implementation, accumulating statistics serves to calibrate metric values against the design organization. After implementation is complete, sufficient data is available to compute variances and confidence levels for complexity values of the finished product. The Halstead, McCabe, or Henry metrics are recommended because they require fewer subjective judgments and have been automated successfully. Many references use a McCabe cyclomatic complexity of less than 10 as indicating an acceptable or good complexity. McCabe reaffirms this in McCabe and Butler 1989.

Even if a solidly managed metric program is in place, caution is advised in interpreting resulting metric values. No existing metric has been thoroughly tested with complex real-time systems, and few metrics, if any, have been tested on multi-program systems. Assigning causes to high correlations is a very chancy endeavor. For example, the Henry information flow metric was validated against higher incidence of module changes during maintenance phase. Does this mean that high information flow complexity means higher error incidence? Consider that high information flow complexity indicates that a module has a greater number of connections with system data. This may mean that the module has a greater chance of being affected by an error (and thus may require modification), but not that the error actually resides within the module. Using a metric outside its range of validation may result in incorrect conclusions.

Some limited use can be made of uncalibrated code or structure metrics. Unfortunately, the most significant results these metrics can provide in these circumstances is bad tidings. There is also a problem with adapting many complexity metrics to practical systems. McCabe and Butler (1989) note that practical systems normally have 100 times the design complexity of those used in textbooks. Low (good) complexity values are desirable, but of limited use unless a lot of ancillary information is available about the development process. All of the recommended complexity metrics (Halstead, McCabe, and Henry) will pinpoint unduly complex modules. The McCabe metric will, in addition, pinpoint unstructured code (McCabe 1976).

Alternatives available to an *a posteriori* reviewer are limited, and complexity metrics can help in making the difficult decision about which alternative to take. A preliminary assessment indicating possible difficulties may provide guidance on where to concentrate investigatory effort. High complexity and unstructured code may support a decision to reject or reverse-engineer a design. Good complexity numbers, coupled with a positive review of the organizational software development process and extensive documentation, may support a decision to proceed.

Although not recommended at this time, two metrics mentioned in this report may be good subjects for further research. These are the application of McCabe's design complexity graphs (McCabe and Butler 1989) and graph-theoretic complexity for architecture (Hall and Preiser 1984). The latter metric was specifically designed for concurrent systems. Regrettably, very little experience has been reported with either metric.

4.3 Recommendations for Future Investigation

To achieve their goals, it is recommended that NRC perform research to develop quantitative metrics for systems and software. The metrics already available for software are a start, but acceptable metric values must be established for NRC needs. The factors and attributes identified for systems and software performance provide an initial list of characteristics that can be considered in developing metrics in these areas.

References

Albrecht, A., November 1983, "Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, Vol. SE-9, 6, pp. 639–648.

Arthur, Jay Lowell, 1985, *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons.

Basili, Victor R., Selby, Richard W., Phillips, Tsai-Yun, November 1983, "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Trans. on Software Engineering*, Vol. SE-9, 6, pp. 652–663.

Boehm, Barry W., 1981, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J.

Boehm, Barry W., Gray, Terence E., and Seewaldt, Thomas, May 1984, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Trans. on Software Engineering*, Vol. SE-10, 3, pp. 290–302.

Brinch Hansen, P., June 1975, "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, Vol. SE-1, 2, 199–207.

British Ministry of Defence, April 1991a, "The Procurement of Safety Critical Software in Defence Equipment Part 1: Guidance," Interim Defence Standard 00-55.

British Ministry of Defence, April 1991b, "The Procurement of Safety Critical Software in Defence Equipment Part 2: Requirements," Interim Defence Standard 00-55.

Brooks, Frederick P., Jr., 1975, *The Mythical Man-Month*, Addison-Wesley Publishing Company.

Burns, Alan, and Wellings, 1990, Andy, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Company.

Fidge, Colin, August 1991, "Logical Time in Distributed Computing Systems," *Computer*, pp. 28–33.

Gorla, Narasimhaiah, Benander, Alan C., and Benander, Barbara A., February 1990, "Debugging Effort Estimation Using Software Metrics," *IEEE Trans. on Software Engineering*, Vol. SE-16, 2, pp. 223–231.

Hall, N.R., and Preiser, S., January 1984, "Combined Network Complexity Measures," *IBM J. Res. Develop. 28*, 1, pp. 15–27.

Halstead, M.H., 1977, *Elements of Software Science*, Elsevier North-Holland, New York.

Henley, Ernest J., Kumamoto, Hiromitsu, 1981, *Reliability Engineering and Risk Assessment*, Prentice-Hall, Englewood Cliffs, N.J.

Henry, Sallie, and Kafura, Dennis, September 1981, "Software Structure Metrics Based on Information Flow," *IEEE Trans. on Software Engineering*, Vol. SE-7, 5, pp. 510–518.

Henry, Sallie, and Selig, Calvin, March 1990, "Predicting Source-Code Complexity at the Design Stage," *IEEE Software*, pp. 36–44.

Hoare, C.A.R., October 1974, "Monitors: An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549–557.

Hoare, C.A.R., August 1978, "Communicating Sequential Processes," *Comm. ACM 21*, 8, pp. 666–677.

Holt, R.C., 1983, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley Publishing Company.

Holt, R.C., Graham, G. S., Lazowska, E. D., Scott, M. A., 1988, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley Publishing Company.

IEEE, 1988a, "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Standard 982.1-1988.

IEEE, 1988b, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Standard 982.2-1988.

Kleinrock, Leonard, and Tobagi, Fouad A., December 1975, "Packet Switching in Radio Channels: Part 1—Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics," *IEEE Trans. on Communications*, Vol. Com-23, 12, pp. 1400–1416.

Kleinrock, Leonard, November 1978, "Principles and Lessons in Packet Communications," *Proc. of the IEEE,* Vol. 66, 11, pp. 1320–1329.

Lehman, M.M., and Belady, L.A., Editors, 1985, "The Characteristics of Large Systems," *Program Evolution—The Process of Software Change, APIC Studies in Data Processing No. 27*, pp. 289–329.

Leveson, Nancy G., Cha, Steven S., Shimeall, Timothy J., July 1991, "Safety Verification of Ada Programs Using Software Fault Trees," *IEEE Software*, pp 48–59.

Lombardo, Thomas G., and Mokhoff, Nicolas, August 1981, "Shuttle Firsts Put to the Test," *IEEE Spectrum*, pp. 34–39.

McCabe, Thomas J., December 1976, "A Complexity Measure," *IEEE Trans. on Software Engineering,* Vol. SE-2¸4 , pp. 308–320.

McCabe, Thomas J., and Butler, Charles W., December 1989, "Design Complexity Measurement and Testing," *Comm. of ACM 32*, 12, pp. 1415–1425.

Metcalfe, Robert M., and Boggs, David R., July 1976, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM 19*, 7, pp. 395–404.

MoD, see British Ministry of Defence.

Muppala, Jogesh K., Woolet, Steven P., and Trivedi, Kishor S., May 1991, "Real-Time-Systems Performance in the Presence of Failures," *Computer*, pp. 37–47.

Nelson, Bruce Jay, May 1981, *Remote Procedure Call*, doctoral dissertation, Carnegie-Mellon University, CMU-CS-81-119.

Parnas, David L., March 1979, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 2, pp. 128–138.

Parnas, David L., December 1985, "Software Aspects of Strategic Defense Systems," *Comm. ACM 28*, 12, pp. 1326–1335.

Preckshot, George G., and Butner, David N., 1987, "A Simple and Efficient Interprocess Communication System for Actually Using a Laboratory Computer Network," *IEEE Trans. Nuclear Science N-34*, 858.

Riezenman, Michael, September 1991, "Revising the Script After Patriot," *IEEE Spectrum*, pp. 49–52.

Rombach, H. Dieter, March 1990, "Design Measurement: Some Lessons Learned," *IEEE Software*, pp. 17–25.

Sahal, Devendra, June, 1976, "System Complexity: Its Conception and Measurement in the Design of Engineering Systems," *IEEE Trans. on Systems, Man, and Cybernetics*, pp. 440–445.

Savitzky, Steven R., 1985, *Real-Time Microprocessor Systems*¸ Van Nostrand Reinhold Company, New York, New York.

Sha, Lui, and Goodenough, John B., April 1990, "Real-Time Scheduling Theory and Ada," *Computer*, pp. 53–62.

Sha, Lui, Rajkumar, Ragunathan, and Lehoczky, John P., September 1990, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, Vol. 39, 9, pp. 1175–1185.

Shoch, John F., and Hupp, Jon A., December 1980, "Measured Performance of an Ethernet Local Network," *Comm. ACM 23*¸ 12, pp. 711 -721.

Smith, Connie U., 1990, *Performance Engineering of Software Systems*, Addison-Wesley Publishing Company.

Wietzman, Cay, 1980, *Distributed Micro/Minicomputer Systems*, Prentice-Hall, Englewood Cliffs, N.J.

Wyman, R.H., et al, December 5–9, 1983, "An Overview of the Data Acquisition and Control System for Plasma Diagnostics on MFTF-B," *Proc. of 10th Symposium on Fusion Engineering*, Philadelphia, PA.

Zhao, Wei, Ramamritham, Krithivasan, and Stankovic, John A., May 1987, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, 5, pp. 564–577.

Glossary

Asynchronous event. An event which is not synchronized with the execution of an instruction in a computer. Typically, something external to the computer that causes an interrupt.

Branch. *IEEE Std 610.12-1990:*

(1) A computer program construct in which one of two or more alternative sets of program statements is selected for execution.

(2) A point in a computer program at which one of two or more alternative sets of program statements is selected for execution.

(3) Any of the alternative sets of program statements in (1).

(4) To perform the selection in (1).

Additional explanation:

In computer machine code, for many computer designs, the default location for finding the next instruction to execute is the next sequential location in memory. An instruction to take the next instruction from a different location, including subroutine jumps and returns, is called a branch instruction. Branch instructions may be conditional upon register values or the results of previous instructions (e.g., arithmetic overflow, parity, sign, etc.). Branches complicate code. Code which has few branches is easier to verify. It also makes fewer decisions.

Branch instructions are generated by compilers in response to conditional statements in the high-level language being compiled. For example, "if," "while," and "do" statements will result in branches.

Completion routine. A completion routine is a subprogram within a program that is executed as a result of an interrupt, but after the interrupt service routine has finished. The completion routine executes within the context of the parent program, having access to parent program data, but with separate program counter and working register values. In UNIX systems, a similar but not identical construct is called a signal handler. Both completion routines and signal handlers are forms of light-weight tasks.

Completion routines preempt parent program code, and so may access the same data items being accessed by parent code. This makes completion routines potentially critical regions, and they are often used to implement synchronization methods by passing data through interrupt-impervious data structures like ring buffers.

Concurrency. A condition where two or more instruction streams execute simultaneously or apparently simultaneously. In single processor systems, apparent concurrency occurs because a lower-priority task may be interrupted by higher-priority tasks, giving the appearance of simultaneous execution. Concurrently executing programs may attempt to access the same data, giving rise to inconsistent data values. Such regions of code (where this is possible) are called critical sections, and considerable effort must be taken to synchronize access to such code. See Hoare 1974 for further details. See Holt et al 1978 or Holt 1983 for texts on concurrent programming.

Context. Context is the entire state of an execution unit, including register contents, floating point register contents, memory mapping state, hardware priority state, and current program counter value. The number of things included in context can vary considerably depending upon Central Processing Unit (CPU) design. Some Reduced Instruction Set Computer (RISC) CPUs, for instance, may save as many as 80 32-bit quantities as context.

Context is important because it is saved and restored during interrupts and when switching between tasks. Sometimes only a limited context save is performed during interrupts. The effort and delay associated with a task switch is justified when tasking is used to modularize and separate the work of one task from another. See Savitzky 1985 for more detail on context switching.

Continuous-time simulation. A method of simulation in which dynamic systems are modeled by differential equations which are solved by various numerical analysis techniques. Time is considered a continuous parameter, and solutions are considered to be analytically continuous between solution points. This differs from discrete event simulation in that discrete event systems are defined only at event times, and represent not dynamic systems but logical systems.

Critical section. "A sequence of statements that must appear to be executed indivisibly is called a critical section. The synchronization required to protect a critical section is known as mutual exclusion" (Burns & Wellings 1990).

Data structure. *IEEE Std 610.12-1990:*

A physical or logical relationship among data elements, designed to support specific data manipulation functions.

Additional explanation:

A usually contiguous section of memory which contains some combination of elementary data types (integers, pointers, floating point numbers) and other data structures which is treated as a single variable for purposes of data modularity and argument passage. A grouping of smaller data items.

Deadlock. *IEEE Std 610.12-1990:*

A situation in which computer processing is suspended because two or more devices or processes are each awaiting resources assigned to the others.

Discrete-event simulation. A type of simulation in which the system under simulation is driven by events which occur at discrete moments in time. The system's response may generate more events which occur at future times, which the simulator captures, queues, and feeds back into the system under test. Discrete event simulators are the underlying form of simulator used for many performance model simulations, petri net simulations, and queuing network simulations.

Embedded system. *IEEE Std 610.12-1990:*

A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.

Additional explanation:

There is also the implication that the computer system is not programmable by its usual users and that it is not easily removable or convertible to another purpose.

Event-based. A system organizing concept (from the programmer's point of view) wherein an application program is notified of outside occurrences by events. Contrast with message-based.

Exception. *IEEE Std 610.12-1990:*

An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception.

Additional explanation:

Interrupts are considered a type of exception, but not the reverse. An exception not only suspends normal program execution, but saves at least the location in program where the exception occurred (so as to return to it later) and restarts the processor at the beginning of exception handling

code. In some designs, exception processing by hardware is quite complex, and a lot of processor context is saved automatically by hardware. In other designs, notably most RISC designs, only the bare minimum of context is saved.

In some languages, Ada for example, exceptions are a language construct which allow software to add exceptions to those already produced by hardware. See synchronous event.

Foreground/background. A simplified form of prioritized multi-tasking consisting of two tasks, the foreground task and the background task. When the foreground task is ready to run it preempts the background task. This is one of the more common arrangements for simple real-time systems.

Function. *IEEE Std 610.12-1990:*

(1) A defined objective or characteristic action of a system or component. For example, a system may have inventory control as its primary function.

(2) A software module that performs a specific action, is invoked by the appearance of its name in an expression, may receive input values, and returns a single value.

Interprocess communication. Communication between processes or tasks which are executing asynchronously with respect to each other, in different contexts, and possibly on different processors. In any interprocess communication there is inherent synchronization in that the sender must send before the receiver receives. Progressively more strict synchronization disciplines may be imposed, e.g., broadcast, one-to-one, rendezvous, remote procedure call.

In light of the definition of process, interprocess communication is probably a misnomer, and should be styled intertask communication.

Interrupt. *IEEE Std 610.12-1990:*

(1) The suspension of a process to handle an event external to the process.

(2) To cause the suspension of a process.

(3) Loosely, an interrupt request.

Additional explanation:

Interrupts are considered exceptions caused by asynchronous events, whereas traps and other exceptions are caused by synchronous events. Like other exceptions, hardware usually saves some portion of context, minimally a return address.

Inter-task communication. Synonymous with interprocess communication.

Kernel. *IEEE Std 610.12-1990:*

(1) That portion of an operating system that is kept in main memory at all times.

(2) A software module that encapsulates an elementary function or functions of a system.

Additional explanation:

A misnomer when applied to the UNIX operating system.

Light-weight task. A thread which can execute in parallel to the main thread of a parent task, while sharing some of the context of the parent task. It is called a light-weight task because only a limited context switch is required to switch between it and its parent.

Message-based. A system organizing concept (from the programmer's point of view) wherein an application program is notified of outside occurrences by the reception of messages. Contrast with event-based.

Multi-tasking. *IEEE Std 610.12-1990:*

A mode of operation in which two or more tasks are executed in an interleaved manner.

Additional explanation:

A term usually applied to an operating system which can execute two or more tasks apparently concurrently (i.e., interleaved). The advantage of this is that each task can follow an external physical process independently of other tasks following other physical processes that have significantly different time histories.

Multi-threaded. Having two or more code threads which can be executed apparently concurrently. Usually applied to code residing within a task.

Petri net. *IEEE Std 610.12-1990:*

An abstract, formal model of information flow, showing static and dynamic properties of a system. A Petri net is usually represented as a graph having two types of nodes (called places and transitions) connected by arcs, and markings (called tokens) indicating dynamic properties.

Polling loop. A program organization technique by which a list of inputs is traversed repeatedly and tested for readiness. Each ready input is dealt with on the spot and then the program proceeds to the next input in list order. This has the advantage of extreme simplicity.

Preemption. Suspending a task (usually because of an interrupt), saving its context, and starting a higher priority task in its place.

Priority. *IEEE Std 610.12-1990:*

The level of importance assigned to an item.

Additional explanation:

An integer ordering placed on tasks which governs a strict rule of preemption. Priority can be static or dynamic. Static priority is fixed at the creation of a task. A task with dynamic priority can have its priority changed by fairness schemes during execution or by inheritance (e.g., it is blocking a higher-priority task due to resource contention.)

Procedure. *IEEE Std 610.12-1990:*

(1) A course of action to be taken to perform a given task.

(2) A written description of a course of action as in (1); for example, a documented test procedure.

(3) A portion of a computer program that is named and that performs a specific action.

Additional explanation:

A procedure is a program or subprogram which performs a sequence of computations or input/output operations. As a subprogram, a procedure is invoked with arguments (and possibly global variable values) and returns results in the same or other arguments (and possibly in global variables). In some high-level languages, a distinction is made between procedures and functions, with functions returning a value and procedures not.

Process. *IEEE Std 610.12-1990:*

(1) A sequence of steps performed for a given purpose; for example, the software development process.

(2) An executable unit managed by an operating system scheduler.

(3) To perform operations on data.

Additional explanation:

Process is sometimes used interchangeably with task, depending upon whose documentation is being read. The IEEE standard (1) defines the word with the majority; process is what is done, task is a computer-mechanized instance of doing it.

Protocol. *IEEE Std 610.12-1990:*

A set of conventions that governs the interaction of processes, devices, and other components within a system.

Additional explanation:

A set of standard formats for encapsulating data sent through a network so that the data can be routed, sequenced, reassembled, transformed, and delivered reliably and correctly. Communication protocols mediate between systems, not within systems.

Real-time system. The IEEE Std 610.12-1990 definition omits the mention of deadlines, which are central to current real-time theory.

Real-time systems are digital systems which must produce correct output in response to input within well-defined deadlines (Burns & Wellings 1990).

(1) Real-time systems have inputs connected to real-world events.

(2) Events (inputs) cause computations to occur which finish before well-defined deadlines, measured from the event.

(3) Computations produce outputs that are connected to the real world.

Software for real-time systems has an additional correctness constraint: even if the values produced by software are correct, the software is considered to fail if it does not meet its deadline. Current-day practice further divides real-time systems into hard real-time systems and soft real-time systems. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and will still function correctly.

Remote procedure call. A restricted rendezvous with the semantics of procedure call (Nelson 1981). The caller does not depart until it has an answer from the callee.

Rendezvous. A synchronization discipline for sending messages between tasks. Both the sender and the receiver arrive (and wait) at a rendezvous until they can simultaneously depart. The sender departs knowing that the receiver received and the receiver departs knowing this also. This has the advantage of providing absolute knowledge to the involved tasks that the rendezvous was successful, but may leave either waiting forever.

Resource. Generally, a resource is any identifiable object or capability of a computer system required to get a job done and that can potentially be shared among two or more jobs. Objects can include devices, portions of memory, or shared variables. Capabilities can include CPU time or datalink capacity. The general meaning is the one applied when adding up or estimating needed resources for performance estimation (Smith 1990).

Formally in a specific computer system implementation, a resource is a named object which is allocated to one or more processes requiring it and released as processes are finished with it (Burns and Wellings 1990). For example, a printer may be allocated to a job printing a file and released at the end of the file. The allocation and release protocol prevents portions of other files from appearing intermixed in the file currently being printed.

Run-time kernel. A kernel generally provided for a real-time embedded system. The implication is that it is fast, has many good real-time features, and provides a basis upon which to build a real-time application.

Scalability. The property that describes the ease of extension of small to large. Scalability includes both the extension of actual small systems to large systems and the application of small system techniques inappropriately to the large system domain.

Shared memory. Physical memory that is attached to two or more processors by multiple ports on the memory module or through a shared bus with hardware contention arbitration. Speed of access is the same or slightly slower than for memory attached exclusively to one processor.

Signal. A UNIX abstraction which sometimes acts like a trap, an exception, or an interrupt.

Software structure chart. *IEEE Std 610.12-1990:*

(1) A diagram that identifies modules, activities, or other entities in a system or computer program and shows how larger or more general entities break down into smaller, more specific entities. Note: The result is not necessarily the same as that shown in a call graph.

(2) Call graph—a diagram that identifies the modules in a system or computer program and shows which modules call one another.

Additional explanation:

For many people, structure chart means call graph.

Synchronization. The term "synchronization" is overloaded in electrical engineering and computer science, which may reflect its importance. To place the term in perspective, its meanings in a variety of contexts are given:

Communications: maintaining phase lock between a multiple of the frequency of a carrier or subcarrier and demodulation circuitry in a receiver.

Computer science/operating systems: queuing an event or data item so that it can be handled later by software that is already busy doing something. When the handler software is ready to service a new request, it asks for the next event or data.

Computer science/distributed systems: maintaining time or event order at physically separated points in a distributed system.

Synchronous event. An event that occurs as a result of executing a low-level (machine language) computer instruction. Typically, this might be an exception due to an instruction fault, memory mapping error, or deliberate system call.

Task. *IEEE Std 610.12-1990:*

(1) A sequence of instructions treated as a basic unit of work by the supervisory program of an operating system.

(2) In software design, a software component that can operate in parallel with other software components.

Additional explanation:

Task also conveys the implication of independence. Tasks usually execute in their own context.

Thread. A sequence of instructions which has no parallel components. The current position in the thread can be represented by a single number (address). Threads are currently a part of the IEEE P1003.4a Posix draft standard for real-time enhancements to portable operating systems.

Trap. *IEEE Std 610.12-1990:*

(1) A conditional jump to an exception or interrupt handling routine, often automatically activated by hardware, with the location from which the jump occurred recorded.

(2) To perform the operation in (1).

Additional explanation:

Trap comes from the expression "trapped instruction," which early computer makers used to implement in software those instructions they had been unable to commit to hardware. Trap is now sometimes used interchangeably with interrupt and exception.